Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentácia k projektu do predmetu IFJ

Prekladač jazyka IFJ17

Tím 053, varianta I

6. decembra 2017

Členovia tímu:
Jakub Filo (xfiloj01) - 25%
Anton Firc (xfirca00) - 25%
Martin Foltýn (xfolty15) - 25%
Jakub Liška (xliska16) - 25%

Obsah

ÚVOD

Dokumentácia popisuje implementáciu prekladača jazyka IFJ17, ktorý je zjednodušenou podmnožinou jazyka FreeBASIC . Jeho implementácia je rozdelená do 4 hlavných častí, ktoré budú bližšie popísané v ďalšej časti dokumentácie:

1. Lexikálna analýza
2. Syntaktická analýza
3. Sémantická analýza
4. Generovanie kódu

Vybrali sme si variantu I. (implementácia tabuľky symbolov pomocou binárneho vyhľadávacieho stromu). Súčasťou dokumentácie sú prílohy, ktoré obsahujú diagram konečného automatu popisujúceho lexikálny analyzátor, LL-gramatiku, LL-tabuľku a precedenčnú tabuľku, ktoré sú jadrom syntaktického analyzátora.

1 IMPLEMENTÁCIA PREKLADAČA JAYZKA IFJ17

1.1 Lexikálna analýza

Lexikálny analyzátor je spracovaný v súbore `tokens.c` a `tokens.h`. Má za úlohu načítať zdrojový súbor a previesť lexémy na tokeny, pričom ignoruje biele znaky a komentáre. Tokenizér funguje ako konečný automat. Funkcia `token_get` spracuje riadok a vracia token, alebo lexikálnu chybu, pokiaľ neexistuje pravidlo pre danú postupnosť znakov. `Token` sa skladá z 2 častí: typu tokenu a samotných dát, ktoré sú uložené v poli znakov.

1.2 Syntaktická analýza

Syntaktická analýza (bez spracovania výrazov)

Syntaktický analyzátor (parser) kontroluje množinu pravidiel, ktorá určuje prípustné konštrukcie daného jazyka. Základné možné konštrukcie jazyka IFJ17 sú definované v LL-gramatike (príloha č. 2). Samotná syntaktická analýza je riešená metódou rekurzívneho zostupu, je jadrom celého prekladača a spolupracuje s ostatnými časťami prekladača. Preklad je riadený syntaxou, pri prechode zdrojovým kódom prebieha syntaktická kontrola a zároveň sa ukladajú získané tokeny do poľa pre neskoršie spracovanie - po získaní tokenu značiaceho koniec riadku (koniec riadku značí v jazyku IFJ17 koniec príkazu). Pokiaľ prebehne syntaktická a sémantická analýza úspešne, sú generované inštrukcie v cieľovom medzikóde IFJcode17. Generovanie kódu pre vstavané funkcie prebieha po spracovaní deklarácií funkcií.

Syntaktická analýza (spracovanie výrazov)

Syntaktická analýza výrazov sa nachádza v súbore `exprs.c` a `exprs.h`. Je implementovaná metódou zdola-hore, pomocou precedenčnej tabuľky(viď. príloha č.3), ktorá obsahuje pravidlá prednosti pre jednotlivé operátory a operandy. Na vstup dostáva syntaktická analýza pole tokenov, ktoré sa ďalej spracúva pomocou 2 zásobníkov pre dátový typ `Token`. Výraz sa postupne redukuje, až na zásobníku zostanú len dva tokeny - $ a E. Po úspešnom priebehu syntaktickej analýzy je výraz prevedený z infixového do postfixového zápisu. Výstup prevodu je ukladaný do lineárneho zoznamu, v ktorom sa výraz ďalej spracúva a postupne prepisuje na inštrukcie.

1.3 Sémantická analzýa

Sémantická analýza zabezpečuje kontrolu logického významu a platnosti jednotlivých výrazov v rámci jazyka IFJ17. Podporuje implicitnú typovú konverziu medzi dátovými typmi `integer` a `double`. Spolupracuje s tabuľkou symbolov pre premenné, v ktorej vyhľadáva pomocou názvu premennej a jej majiteľa[1]. Pridanie informácie o majiteľovi premennej zabezpečuje možnosť deklarácie premenných s rovnakým názvom v rôznych kontextoch (hlavné telo programu, telo funkcie). Taktiež využíva tabuľku symbolov pre funkcie, v ktorej je možné vyhľadávať pomocou názvu(identifikátora) alebo názvu označenia[2]. Pri volaní funkcie prebieha kontrola počtu parametrov a takisto kontrola dátových typov jednotlivých parametrov.

[1] Majiteľ značí, v akom kontexte sa premenná nachádza.
[2] label_name

1.4 Tabuľka symbolov

Vybrali sme si variantu I. ktorá zahŕňa implementáciu tabuľky symbolov pomocou binárneho vyhľadávacieho stromu. Prekladač používa dve tabuľky symbolov, jednu pre identifikátory premenných a druhú pre identifikátory funkcií. Tabuľka symbolov je impementovaná v súboroch `symtable.c` a `symtable.h`. Pri implementácii boli využité alogoritmy prednášané v kurze IAL.

1.5 Generovanie kódu

Generovanie kódu je implementované v súboroch `istructions.c` a `istructions.h`. Po úspešnom priebehu syntaktikej a sémantickej analýzy sú volané príslušné funkcie pre generovanie kódu podľa významu danej sekvencie tokenov. Pre každú inštrukciu existuje funkcia ktorá volá hlavnú funkciu `instruction` ktorá zabezpečuje finálny zápis inštrukcie. Funckia `instruction` prijíma ako parametre premenlivý počet reťazcov znakov, ktoré popisujú názov inštrukcie a parametre. Všetky reťazce znakov prijaté ako parametre sú nakoniec zapísané ako jeden reťazec znakov(názov inštrukcie a jej parametre) na štandardný výstup.

1.6 Ostatné Dátové Štruktúry

Token

Token je definovaný pre každú lexikálnu jednotku, obsahuje typ tokenu a samotné dáta. Tvorí základnú jednotku pre predávanie informácií medzi lexikálnym analyzátorom a syntaktickým analyzátorom.

Variable

Variable je štruktúra obsahujúca informácie o jednej premennej. Dôležité informácie identifikujúce premennú sú jej názov, dátový typ a majiteľ (používa sa pri vyhľadávaní).

Function

Function je štruktúra obsahujúca informácie o jednej funkcii. Dôležité informácie identifikujúce funkciu sú jej názov, názov označenia[3] a parametre funkcie.

Zásobník

Prekladač využíva viac zásobníkov, podľa typu dát, ktoré je potreba uložiť.
Zásobník pre dátový typ `Token` je použitý pri syntaktikej kontrole a vyhodnocovaní výrazu.
Zásobník pre dátovy typ `Context` zaisťuje udržiavanie kontextu, v ktorom sa práve prekladač nachádza (hlavné telo programu, telo funkcie).
Zásobník pre dátový typ `Void` ukladá adresy pamäti alokovanej za behu programu. Pamäť je uvoľnovaná naraz, buď pri úspešnom ukončení prekladu, alebo pri chybe.

[3] label_name

Obojsmerne viazaný lineárny zoznam

Zoznam je použitý pre uloženie výrazu v postfixovom tvare a následnú prácu s ním. Do zoznamu sú ukladané tokeny, reprezentujúce jednotlivé časti výrazov. Tie sú postupne spracovávané a nahradzované názvami pomocných premenných.

Binárny vyhľadávací strom

Implementuje tabuľku symbolov viď sekciu 1.4.

2 PRÁCA V TÍME

Prácu v tíme sme si rozdelili nasledovne:

- **Jakub Filo**: Syntaktická analýza(spracovanie výrazu), tvorba precedenčnej tabuľky, vyhodnocovanie výrazov, testovanie, dokumentácia
- **Anton Firc**: Syntaktická analýza(bez spracovania výrazu), tvorba LL gramatiky, vstavané funkcie, testovanie, dokumentácia
- **Martin Foltýn**: Lexikálna analýza, parser, generátor inštrukcií, testovanie, dokumentácia
- **Jakub Liška**: Vstavané funkcie, generátor inštrukcií, správa pamäte, chybové hlásenia, testovanie, dokumentácia

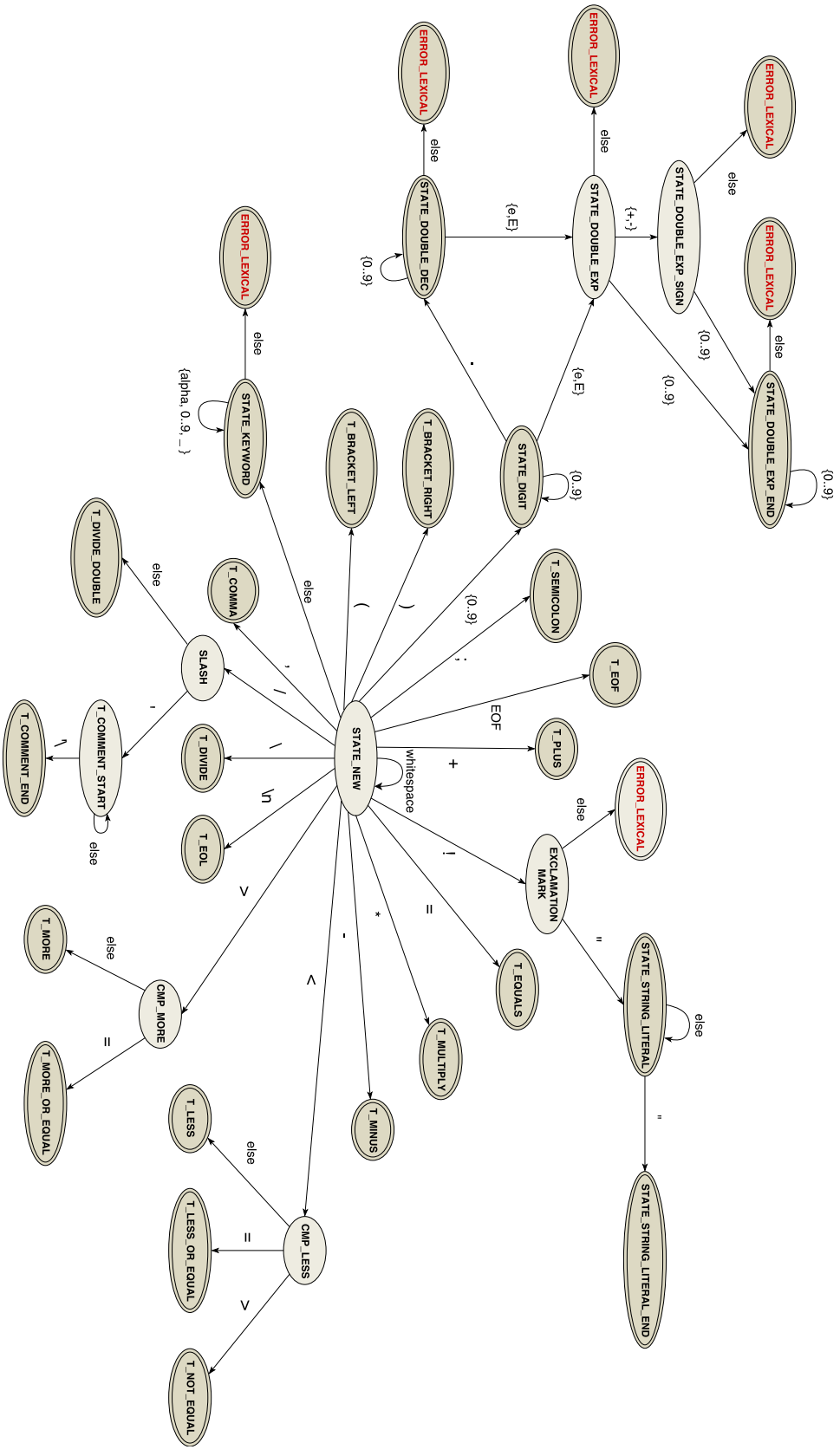
Komunikácia v tíme prebiehala z väčšiny osobne, na rozdeľovanie úloh sme používali Trello.

3 ZÁVER

Tento projekt bol podľa všetkých členov tímu najkomplexnejší s akým sme sa počas štúdia stretli. Vďaka nemu sme sa zdokonalili v jazyku C, používaní GITu, ale hlavne v tímovej spolupráci a s tým súvisiacej organizácii času a úloh. Takisto sme získali nové skúsenosti s vývojom rozsiahlejšieho softvéru, ktoré určite využijeme v praxi pre zrýchlenie a zjednodušenie procesu vývoja. Výhodou bolo, že bývame vo vedľajších izbách, a preto sme mohli všetky problémy a prípadné nejasnosti riešiť osobne a okamžite. Prenikli sme do problematiky prekladačov a o to viac si vážime programátorov, ktorí ich vytvárajú.

4 PRÍLOHY

4.1 Diagram konečného automatu



4.2 LL-gramatika

1. <prog> -> <functions_declaration> <functions_definition> SCOPE EOL <stat_list> END SCOPE
2. <functions_declaration> -> <function_declaration> EOL <functions_declaration>
3. <functions_declaration> -> E
4. <function_declaration> -> DECLARE FUNCTION ID (<function_arguments>) AS VAR_TYPE EOL
5. <functions_definition> -> <function_definition> EOL <functions_definition>
6. <functions_definition> -> E
7. <function_definition> -> FUNCTION ID (<function_arguments>) AS VAR_TYPE EOL <stat_list> <is_return> END FUNCTION
8. <stat_list> -> <stat> EOL <stat_list>
9. <stat_list> -> E
10. <stat> -> DIM ID AS VAR_TYPE <var_def> EOL
11. <var_def> -> E
12. <var_def> -> = EXPR
13. <function_arguments> -> <function_argument> <function_arguments_n>
14. <function_arguments> -> E
15. <function_argument> -> ID AS VAR_TYPE
16. <function_arguments_n> -> E
17. <function_arguments_n> -> , <function_argument> <function_arguments_n>
18. <stat> -> ID <what_id>
19. <id_assign> -> EXPR
20. <id_assign> -> ID (<function_arguments_call)
21. <stat> -> PRINT <exprs> EOL
22. <exprs> -> EXPR <exprs_n>
23. <exprs_n> -> ; EXPR <exprs_n>
24. <exprs_n> -> E
25. <stat> -> IF EXPR THEN EOL <stat_list> ELSE EOL <stat_list> END IF EOL
26. <stat> -> DO WHILE EXPR EOL <stat_list> LOOP EOL
27. <stat> -> INPUT ID EOL
28. <stat> -> EOL
29. <is_return> -> RETURN EXPR EOL

30. <is_return> -> E
31. <what_id> -> = <id_assign> EOL
32. <what_id> -> (<function_arguments_call>) EOL
33. <function_arguments_call> -> <function_argument_call> <function_arguments_call_n>
34. <function_arguments_call> -> E
35. <function_argument_call> -> EXPR
36. <function_arguments_call_n> -> E
37. <function_arguments_call_n> -> , <function_argument_call> <function_arguments_call_next>

LL-tabuľka

	SCOPE	DECLARE	FUNCTION	EOL	DIM	AS	ID	(	)	,	RETURN	END	IF	THEN	ELSE	DO	=	INPUT	PRINT	LOOP	EXPR	;
<prog>		1																				
<functions_declaration>	3	2	3																			
<function_declaration>		4																				
<functions_definiton>	6		5																			
<function_definition>			7																			
<stat_list>				8	8		8				9	9	8		9	8		8	8	9		
<stat>				28	10		18						25			26		27	21			
<var_def>				11													12					
<function_arguments>							13		14													
<function_arguments_n>									16	17												
<function_argument>							15															
<id_assign>							20														19	
<exprs>																					22	
<exprs_n>				24																		23
<is_return>											29	30										
<what_id>								32									31				31	
<function_arguments_call>							33		34													
<function_arguments_call_n>									36	37												
<function_argument_call>																					35	

4.3 Precedenčná Tabuľka

	+	-	*	\	/	(	)	=	<>	<	>	<=	>=	$	i
+	>	>	<	<	<	<	>	>	>	>	>	<	>	>	<
-	>	>	<	<	<	<	>	>	>	>	>	<	>	>	<
*	>	>	>	>	>	<	>	>	>	>	>	>	>	>	<
\	>	>	<	>	<	<	>	>	>	>	>	>	>	>	<
/	>	>	>	>	>	<	>	>	>	>	>	>	>	>	<
(	<	<	<	<	<	<	=	<	<	<	<	<	<	>	<
)	>	>	>	>	>	x	>	>	>	>	>	>	>	x	x
=	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<
<>	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<
<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<
>	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<
<=	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<
>=	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<
$	<	<	<	<	<	<	x	<	<	<	<	<	<	x	<
i	>	>	>	>	>	x	>	>	>	>	>	>	>	>	x

1. `<E> -> <E> + <E>`
2. `<E> -> <E> - <E>`
3. `<E> -> <E> * <E>`
4. `<E> -> <E> / <E>`
5. `<E> -> <E> \ <E>`
6. `<E> -> <E> < <E>`
7. `<E> -> <E> > <E>`
8. `<E> -> <E> <= <E>`
9. `<E> -> <E> >= <E>`
10. `<E> -> <E> = <E>`
11. `<E> -> <E> <> <E>`
12. `<E> -> (<E>)`
13. `<E> -> integer`
14. `<E> -> double`
15. `<E> -> string`
16. `<E> -> id`